The R Researcher's companion v. 0.01

A gentle introduction to R with applications in war and peace

David Randahl

2022-11-02

Contents

Preface

Welcome to this draft online edition of *The R Researcher's companion.* This is a project that has been in the works for a long time and is finally, hopefully, coming together. In this 'book' (really, it is just a collection lecture and workshop notes) my ambition is to convince you that R is not an obstacle to be overcome. Rather, I want to convince you that knowing R is a gateway to a fantastic new world of data analysis and exploration. I want to convince you that knowing R will open up new doors for you to understand your data, to understand your methods, and to answer research questions in ways you previously thought not possible. While learning R may seem (unnecessarily) complicated, I assure you that learning enough R to perform basic data analysis is within the grasp of anyone.

Why another book about R?

This book is born out of half a decade of teaching R to undergraduate and graduate students in the social sciences. Teaching R in this context is always challenging as many of the students are neither interested in R itself, nor the underlying statistical theory which underpins modern data analysis and on-top of which R is built. Instead, these students are interested in the output of the data analysis, the plots, diagrams, and charts which describe the data, or the regression tables, correlations, and diagnostics from which inferences about the data can be drawn. The primary challenges for teaching R in this environment are to convince the students that R is not (much) more challenging to learn than alternatives such as SPSS or STATA (or, god forbid, Excel!) which allow them to obtain these outputs without having to rely on coding, and that learning R will actually help them better understand these outputs as they will gain a deeper understanding of the underlying data and the way that it is being processed. Compounding these challenges have been a lack of literature specifically tailored to this category of students, i.e. students with limited or no background in statistics or programming but who are keen on learning how to conduct research using R. The fundamental problem is that most books on research methodology and R fall in one of two camps, (1) books focusing on research methodology and/or data analysis with applications in R or another statistical software but without the fundamentals of R, or (2) books which are focused on data science and R from a programming perspective. Books in the first category are usually very useful for teaching the methodology and how to conduct the analysis itself, but without a strong foundation in R students may feel lost in the applications and may simply resort to copying and pasting code from the book without understanding how and why to edit this code to suit their own research questions. Books in the second category, on the other hand, are generally tailored to a different audience with a stronger interest in the *programming* side of R and who may have a more extensive background in data science and/or programming.

The book you are currently reading aims to fill this gap in the teaching literature for R. This book is supposed to be used as a *researcher's companion*, i.e. a book which you can go back to at any point during your research process to look up how certain concepts work in R and how to figure out where you've gone wrong. It assumes neither any knowledge of R beforehand, not any extensive knowledge of mathematical or statistical methodology. My hope is that this short book will to allow you to go from zero knowledge in R to being comfortable reading and writing your own code, and that it will be a support for you when doing the data analysis in R for your research projects.

What this book will teach you (and what it will not)

As this book is supposed to be a researcher's companion rather than a book on research methodology, there are some strict limitations on what this book will teach you and what it will not. This book will be strictly focused on R, the central concepts you need to understand in order to do data analysis in R, the fundamental pre-processing needed to do data analysis properly, and how to best present your results. On the other hand, this book will *not* teach you research methodology. It will *not* provide you with any fundamental understanding of data analysis techniques beyond the mechanical understanding of how to run some simpler techniques, and it will *not* help you understand diagnostics for methods or when and why to use certain types of methods over other types of methods. Thus, this book is a *companion* intended to be read alongside books which dive deeper into the substantive topics on how to conduct research and best utilize statistical models but which do not help you figure out why you're getting a specific type of error when you're running your code, how to best transform, merge, or otherwise wrangle your data, or how to make stylish and beautiful plots and tables, and how to customize them. This book is not intended to be an exhaustive guide to R, but rather as a introduction to take you over the most common hurdles you will face when embarking on your journey to become proficient in R. Once you've reached the end of this book, you should feel comfortable moving on to more advanced books on programming in R such as Hadley Wickham's fantastic books R for data science and Advanced R or Måns Thulin's excellent applied book Modern Statistics with R.

Outline of the book and how to read it

This book is comprised of three main parts. In the first part, we will focus on the *basics* of R including an introduction to what R is, what the difference between R and RStudio (and other IDEs) are and how a basic workflow in RStudio can look. This part is the most mechanical of the book and will introduce a lot of programming concepts such as functions, objects, and packages which are crucial in order to understand how to work with R. It will also briefly touch upon different types of data and how generic methods differ depending on these data types.

The focus of the second part of the book is *data*. In this part, you will learn how to load different types of data into R, how to transform data by modifying or adding new variables to your data, and how to filter out and select parts of the data. It will also teach you how to combine data from different sources into a single data frame, how to aggregate data across different groups, and how to summarize your data. We will do all of this using the syntax (i.e. way of writing code) of the `tidyverse` package and we will build a toolbox of tools you'll be able to apply whenever you embark on a data analysis project.

In the final part of the book we will look at *data presentation*. Here you will learn how to run simple statistical models, how to make beautiful and customizable plots to visualize your data and your results using the `ggplot` package, and nice-looking well-formatted regression tables using the `stargazer` package. We will also briefly look at some alternatives for making tables using a combination of the `tidyverse` syntax in the `gtsummary` package.

Now without further ado, let's dive into the wonderful world of R!

Part I

Workshop 1: Introduction and basics of R

Chapter 1

What is R and why should you learn it?

1.1 R, a programming language for data analysis

So, what is R really? Well, R is a programming language which was developed as a free open source successor to the *S* programming language and first saw the light of day in the early 1990s'. However, R is not like other programming languages which are often times built around being able to build general purpose applications for use in computer programs, apps, and websites. Instead, R is a *statistical* programming language, built by statisticians and data scientists specifically for the purpose of being a tool for data handling, data presentation, and data analysis.

The fact that R is a programming language allows you to build your own tools or use tools others have built and made available in the over 17,000 unique packages for R in order to best analyze your data and provide answers to your research questions. This flexibility makes R a highly versatile and powerful tool for conducting your analysis compared to alternative data analysis *software* such as SPSS, STATA, SAS, and Excel, which locks the user into the tools available in those specific software. On top of that, R allows you to work directly with the data in a way that no competitor can, it allows you to deep dive into analyses of sub-samples and sub-groups, to merge, transform, and otherwise wrangle data in ways which in other software or programming languages are complicated at best, all while keeping the syntax and workflow neat and tidy. In essence, R is the best tool available for handling and analyzing data currently available.

1.2 Why learn R?

You might think to yourself: *I'm not interested in programming and I just want to be able to conduct some basic data analysis. Why should I learn R?* Great question! Apart from the obvious reason mentioned above, that R is the best tool available for handling and analyzing data, there are several other reasons one should learn R, and the fact that R is free, there are several good reasons why you should invest the time in learning R. First and foremost among these is that learning R will help you better understand the data you are working with and the methods you are trying to use to analyze this data. Being able to easily dive into the data by filtering, selecting, summarizing, and aggregating the raw data along the way brings you *closer* to the data and allows you to better understand the strengths and weaknesses of your data in a way which is neigh on impossible without a flexible tool like R.

R also offers you better opportunities to learn about the methods you are using. If you get a weird or unexpected result when doing the analysis, R offers you the tools to figure out what is happening and whether the results are an effect of something going wrong in the analysis, or if there are quirks in the data which generate these odd results. By working in R you are also promoting the ideal of reproducible research. Since R is open source and free, sharing the R code for your analysis with colleagues or reviewers allows them to see what you have done and thereby easier replicate your results or build on the results when conducting their own research.

The flexibility of R and the ability to work outside the box also tend to generate new types of research questions, as your mind is free to look for interesting phenomena outside the ones which are possible to analyze within a strict framework of already existing models. This is particularly true when you (at a later point in your journey into research methodology!) start to understand what the limits of the most common statistical methods are and you want to move past these to address new types of questions which may arise.

Finally, knowing R is a highly marketable skill, and you will never regret having learned the basics in R even if you end up in a job where this is not used. We are inexorably moving towards a more interconnected world which produces reams upon reams of data about both you and the world you live in, which will inevitably be used in different types of analysis. Having a basic understanding of how such analysis may be done is therefore fast becoming somewhat of a life skill and something which everyone ought to have.

1.3 How to approach learning R

Now that you're convinced of the importance of learning R, how should you approach this topic? R is a programming *language* and learning R is therefore similar to learning a language. However, instead of concepts such as "adjectives," "verbs," and "nouns," you will have to learn concepts such as "objects," "functions," and "arguments." At times it will be frustrating, and at times you will just want to give up. Just remember that as with learning any language, practice is the key to succeeding. However, there are also some key differences in learning a *programming* language compared to a human language.

Primary among these are that when you are communicating in a human language in which you are not fully fluent, the receiver of the communication is usually able to interpret what you're saying despite potential grammatical or other linguistic mistakes and you may therefore make yourself understood despite lacking some of the fundamental understanding of the language. In programming languages, on the other hand, the interpreter is completely unable to understand you if you make any errors in your syntax (akin to grammar in human languages) or if you have any spelling errors in your code. Thus, while it may be frustrating to start learning R by learning things like the equivalent of "Hi, my name is X" or the duolingo-style "The shark did not eat the helicopter" when all you want to do is dive into deeper conversations about the philosophy of science, it is absolutely essential to build a solid foundation of understanding the structure of the language before moving on to the more advanced topics.

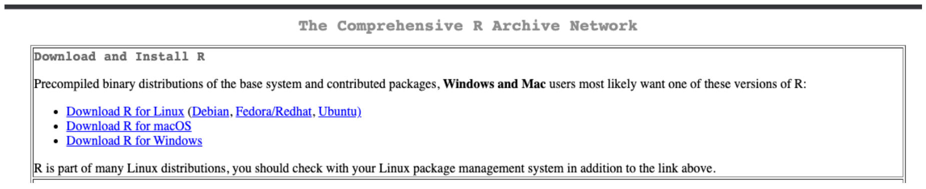
Another difference between human languages and R is that while R, as all programming languages, is completely inflexible when it comes to the syntax and spelling, it also very flexible and has very few opinions on what is the 'right' or 'wrong' way of doing things. Thus, as long as you use correct syntax there are usually a multitude of different ways of doing the same thing, of which neither is necessarily more right or wrong. This feature of R may sometimes be confusing, but it also allows you to use R in the way that is most intuitive to you. It also allows you to develop your specific way of writing R over time and constantly improve the way you write your code for your data analysis.

As with any language, however, the only real way of learning it is by using it. In order to learn R properly you therefore need to find useful projects on which you can use the skills you learn here. Maybe you have some data associated with a course paper or working paper you want to dive deeper into and explore, or maybe you want to use the trial data sets provided with this book to investigate certain relationships. My recommendation would therefore be to find a project or find some data which you are interested in exploring, and keep it near at hand when you are reading this book. The code snippets that will be provided in this book can be seen as phrases from an old-school phrase-book. They are not useful if you only repeat them for yourself, they become useful when you apply them to your own data and adapt them to the specific situation you are in.

Chapter 2

R, RStudio, and how to get started

It is finally time to get started in R. To get started you first need to download and install R. You do this by going to the Comprehensive R Archive Network (CRAN) website, https://cran.r-project.org/. At the top of the website there is a box where you see three links allowing you do download R for your operating system (Linux, macOS, or Windows).


The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux (Debian, Fedora/Redhat, Ubuntu)
- Download R for macOS
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

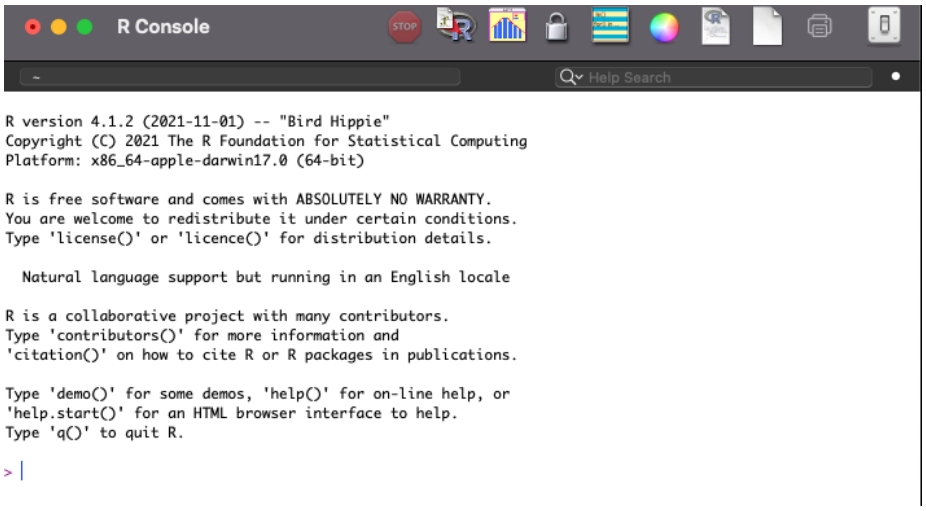
Figure 2.1: Select your operating system to get to the appropriate download site

Follow the link for your operating system and download the latest version of R (at the time of writing R 4.2.1).

If you are on Windows, you should get the 'base' version at the top of the download site. If you are on macOS and have a new M1 mac you should get the version with 'arm64' in its name. If you have an older Intel Mac you get the version without 'arm64' in its name. Once you've downloaded the R install file (.pkg on mac and .exe on windows) install R by opening this file and following the instructions. Congratulations, you are now able to use R!

2.1 RStudio and other IDEs

You've now installed R and could therefore start using it. However, what you have installed is just an interpreter for the R language, nothing else. If you were to open up R as you just had installed you would get to the R console which looks like this.



```
R version 4.1.2 (2021-11-01) -- "Bird Hippie"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

Figure 2.2: The R Console

You are able to run R code in the R console, but it is not very hospitable as it does not by itself keep track of everything we have inside our R session, the code we have run, and the scripts we want to run. We are therefore also going to install RStudio which will help us use R by providing us with a nicer interface with which to interact with R. RStudio is the most popular IDE (integrated developer environment), i.e. interface, for R, and provides a lot of neat functionality for R which will come in handy as we continue to explore R. RStudio is, however, not the only IDE for R, there are a variety of other IDEs which you could potentially use and do not feel locked into RStudio if you find one you like better. Everything we learn in this course is translatable across different IDEs for R.

When thinking about R and RStudio, it is important to distinguish between the language R, underlying interpreter of R, and the interface through which we interact with R. RStudio is not R, rather it is a tool that helps us communicate with R by providing additional features which are helpful in this endeavor. Thus, whenever we work in R, what we actually are doing is working *with* R through RStudio, and RStudio is the program we are going to have open as we continue to work through this book.

2.1.1 Installing RStudio

To start working in RStudio we first need to download and install it. You do this by going to RStudio's website, https://www.rstudio.com/. At the top of the website you can see the 'download' button which will take you to the download page.

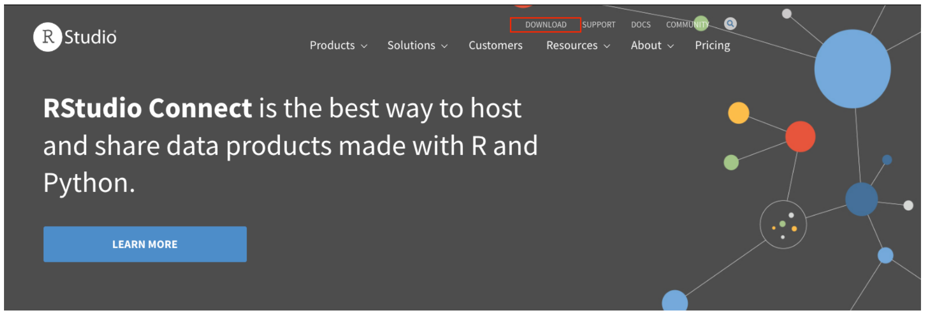


Figure 2.3: The RStudio website

On the download you select the free RStudio Desktop version and click download. This will take you to a second page where you get to select your operating system (Linux, macOS, or Windows). Select the file for your operating system and download it.Once you've downloaded the RStudio install file (.dmg on mac and .exe on windows) install RStudio by opening this file and following the instructions. You are now ready to dive into using R with RStudio!.

Chapter 3

Basic workflow in R using RStudio

When you first open RStudio you will be greeted by a window with three different *panes* (i.e. sub-windows).

[three pane figure]

These three panes are 1. The *console* at the top left. This is where you can interactively run code. When you start R studio you will always see the R start-up message here which gives some basic information about which R version you are using and miscellaneous information about under what licence it may be used, and how to cite, etc. We will use the console in a bit to run our first lines of code. 2. At the top right, we have the *environment* pane. This is where you will see all of your objects, data, and functions (see next chapter) that you have available in your R session. 3. The *files, plots, and help* pane. In this pane you can view all of the files that are in your working directory, plots that you have made, what packages you have installed, and the help files for different functions. Don't worry if you do not know what a working directory, package, or help file is. We will cover this in due time.

Each of the panes have different tabs which you can flip through to see other things such as the history, terminal, and other things. You can also customize the positions of the panes, as well as which tabs go in which pane in the preferences of RStudio.

Missing from these panes is, however, the most important pane, the *script* pane. We can open up the script pane by opening a new script. We do this by pressing the File menu at the top of RStudio and then *'New File > R Script'*. Doing this will open up an empty R script at the top left of the RStudio window, and will move the console pane to the lower left corner of the RStudio window. (Note:

if you opened RStudio by opening an R script file you will already have this fourth window when RStudio starts)

[open script figure]

It is here, in the *script* that we will do most of our work. A script is basically exactly what it sounds like. It is a script of instructions (code) that will be executed in the order that it is written. We will primarily work with scripts rather than directly in the console, as working in a script allows us to keep track of everything that we have done so far.

3.1 Running code

Let us immediately deviate from the rule above and run some code in the console. In practice, everything that we do in R revolves around code. However, do not let that word *code* scare you. Coding is actually much more benign than it may first seem. For instance, we can use R as a calculator by entering `1+1` in the console and pressing enter. When you do this you should immediately see the console spitting out `[1] 2`. And just like that, you've run your first piece of code, 1+1=2! Thus, running code is simply giving R an instruction which it will execute and give an answer.

3.1.1 Scripts

Now, we ran our first piece of code in the console, but as I said previously, most often we will primarily use the script. Running code from the script is slightly different than running it in the console, primarily because the code will not execute when you press enter. Instead you will just go to the next line of the script when you press enter. To execute code in a script you highlight the parts of the script you wish to run and either press Ctrl+Enter (Cmd+Enter on Macs) or the small bottom at the top of the script window called 'Run.' Let's try it by writing `2+2` on the first line in the script, highlighting it, and pressing Ctrl+Enter. When you do this you will see that the code is actually run in the console and you should see

```
> 2+2
[1] 4
```

in the console. This is because even when we write the code in the script, it is still executed in the console. However, as you can clearly see too, executing a piece of a script does not change the script. The scrip thus remains constant, and if you keep writing your code in the script you will have a perfect record of what you have done. A nice thing about scripts is that you can also run multiple lines of code by highlighting a larger section of code. For instance, if we were to add 2*4 to the second line of the script and 3+8*2 to the third line of the script so that our script looks like this

```
1+1
2*4
3+8*2
```

we can run all of these lines of code by highlighting all three lines and pressing Ctrl+Enter. This will execute the code one line at a time in the console, and your output should look like this:

```
> 1+1
[1] 2
> 2*4
[1] 8
> 3+8*2
[1] 19
```

The point of this exercise is of course not that we should use R as a calculator, but rather to illustrate that running code is not something advanced. It is just an instruction to R to do something.

3.1.2 Commenting in a script

When writing a script, you may want to keep track of what you are doing on the individual lines and why you are doing it. You can do this by including comments in the R script, which you do with the # sign. In the case of using R as a calculator, this may of course seem silly, but once you start doing more advanced data processing, it may be super useful to know what each line of code does and why. This is especially true when you are re-visiting an old R script that you have not worked with in a long time and you are trying to figure out what you did and why.

To include a comment, simply write # after any piece of code you wish to comment on. Anything that follows the # on that line will not be interpreted as code and will not be executed by R. For instance we may add the following comments to our code above:

```
1+1 # I want to add one and one
2*4 # This line multiplies two by four
3+8*2 # This line shows that you can do multiple operations in one line
```

When we execute this code, you can see that the comments are still displayed in the console, but the results are exactly as before:

```
> 1+1 # I want to add one and one
[1] 2
> 2*4 # This line multiplies two by four
[1] 8
> 3+8*2 # This line shows that you can do multiple operations in one line
[1] 19
```

3.1.3 Keeping track of our work and saving scripts

The point of having a script is to keep track of what we have done thus far, and to be able to replicate the things that we have already done. Thus, whenever we have worked on a project we should take care to save our scripts. You can save your script by pressing Ctrl+s (cmd+S on a mac), or go to menu at the top and press *'File > Save'* to save your R script. Don't forget to save your script often.

Chapter 4

Objects, functions, and packages

Now that we have a basic understanding about what it means to run code and what scripts are, let us introduce the fundamental workhorses of R. Objects, functions, and packages.

4.1 Objects

An object is simply something, anything, that we want R to store so that we can access it. It can be anything from a single value, to a collection of values, to a data set, a function, or even a list of other objects. The only limit for objects are basically that each object needs to have a unique name, which is used to call the object.

To create an object we use the so called assignment operator, which is simply this arrow: `<-` where we put the name of the object that we want to create on the left hand side of the arrow, and the object itself on the right hand side of the arrow. The name of the object cannot start with a number and is not allowed to contain spaces, or a number of special signs such as `+, -, :, $` and others. A good convention to get around the fact that you are not allowed to use spaces in names is to use either snake_case, in which you separate words with underscores _ and which is the convention I will follow in this book, CamelCase, in which you put a capital letter as the first letter in each word, or period.case, in which you put periods between words. Another good convention is to name your objects descriptively so that you remember what they refer to. Using what we know, we can now create our first object by running the following code. Remember that you should always keep track of what you are doing, so you should write the code in the script and then running it by highlighting it and pressing Ctrl+Enter.

```
my_first_object <- 32
```

If you look at the environment pane, you will now see that an object appeared in our environment, and that object is called 'my_first_object.' The name here is arbitrary, and we could have named it anything (as long as it does not contain spaces or begin with a number). We can now call our object by simply writing the object name and running the code:

```
> my_first_object
[1] 32
```

As you can see, R simply fetches the object and prints it out. As 'my_first_object' is a number, we can treat it as any other number and perform mathematical operation on it, such as running `my_first_object*2+4` which would yield:

```
> my_first_object*2+4
[1] 68
```

We can also define a second object, and assign a number to it. For instance

```
another_object <- 21
```

and then we can call both objects to for instance add them together

```
> my_first_object + another_object
[1] 53
```

If we then want to replace either of the variables, we can simply overwrite them by creating a new object with the same name. For instance by running

```
another_object <- 15
```

Then when we add the objects together we will see that

```
> my_first_object + another_object
[1] 48
```

So, how is this useful in practice? Well, let's say that you are interested in the number of people who have voted for a certain party in an election, but you only have access to the total number of people who voted and the vote shares of each party. We could then for instance run

```
# Calculating the number of votes for a specific party

voting_pop <- 9321800 # The number of people who cast their votes
vote_share <- 0.52 # The vote share of the party of interest

number_of_votes <- voting_pop*vote_share # Number of votes cast for the party
```

If we then want to investigate the number of votes for another party, all we need to do is simply change the value for the object `vote_share` to the vote share of the other party we are interested in knowing the number of votes for. If we are doing many calculations using a single object, we can save a lot of time by structuring our calculations in a script like above.

Another alternative if we want to investigate how many people voted for a number of different parties is to create a *vector* of values containing the vote shares of each party we are interested in. A vector is simply a collection of values, and are created with the function `c()`. We will talk more about functions very shortly, but for now it is enough to know that `c()` is a function which takes multiple values, separated by commas, and combines them into one object. Let us run

```
vote_shares <- c(0.52,0.44,0.04) # Vote shares of all parties of interest
```

We can then access each individual values in the `vote_shares` vector by using hard brackets `[]` with the position inside the brackets. You can try this by running

```
vote_shares[1]
vote_shares[2]
vote_shares[3]
```

Similarly we can obtain the number of people voting for each party by multiplying the vote shares from each party with the total number of voters

```
voting_pop*vote_shares[1] # Number of people who voted for the first party
voting_pop*vote_shares[2] # Number of people who voted for the second party
voting_pop*vote_shares[3] # Number of people who voted for the third party
```

4.2 Functions

While objects are everything that we store in R, functions are everything that we *do* in R. Functions make the calculations, and functions are what makes R a statistical programming language. A function is a piece of code that takes some object or objects and performs some operations on them. For instance, we could take the mean or sum of some values, we can run a regression, or we can make simulations about which countries are likely to descend into civil wars. Functions can be recognized by the fact that they are almost always followed by a parenthesis. Within this parenthesis we will put the arguments of the function. An argument can be thought of as a direction or an instruction to the function. A function may have more arguments than just one, and in that case we separate the arguments with commas. These other arguments may be instructions such as “remove missing values before doing the calculations,” or “make the points in this plot blue,” or any other kind of instruction which the function allows. Functions in R are objects as well, and called just like other objects by writing their name and follow the form `function_name(argument_name)`. We can try this out by applying the `sum()` function to our vector of vote shares.

```
sum(vote_shares)
```

In this case, `sum()` is the function we are calling and our object `vote_shares` is the first (and only) argument. As we can see, the vote shares in our vector sums to one.

Now, let us expand our example a bit and let us say that there are 10 electoral districts of different sizes and we have access to both the number of voters in each districts, and the vote shares for the first party in each district. We can enter these into R by running

```
voters_per_district <- c(618880, 1286117, 318003, 1037879, 505025, 493486, 1621599, 97
vote_share_per_district <- c(0.506, 0.583, 0.618, 0.445, 0.219, 0.461, 0.680, 0.280, 0
```

We can now make some calculations on these vectors. For instance, by running `mean(vote_share_per_district)` we can see that the mean vote share across the districts for the first party was approximately 49.4%, that is, a bit lower than their total vote share of 52%. How is this possible? Well, the electoral districts are of varying sizes, and thus since the total vote share is higher than the mean vote share, this would indicate that the party got a higher vote share in the larger districts. We can see whether or not this holds true by checking the correlation between the two vectors. We do this with the function `cor()`. `cor()` can give us the correlation between two different vectors if we input the two vectors as the first and second arguments of the function. Thus, to calculate this correlation, we simply run `cor(vote_share_per_district,voters_per_district)`. As you can see we separate the two arguments `vote_share_per_district` and `voters_per_district` by a comma, and then we get the result `0.42695` indicating a positive correlation between the two vectors and thus that the party got a higher vote share in the larger districts. Whether or not this correlation is substantive, large, or important is a different question and not something we will dive into here.

4.2.1 Default arugments and named arguments

If you have worked with correlations before you will know that there are different types of correlations such as pearson's, spearman's, and kendall's correlations. So which correlation is it that we have calculated above? And how would we know? R can handle all three of the correlations above, and all of them within the regular `cor()` function. The reason that we can run the `cor()` function without specifying which type of correlation we want is that R has a system of *default arguments*, i.e. values which the argument will take *unless* an other value is specified. In the case of the `cor()` function, pearson's correlation is the default and thus the one which is calculated above. If we want to use another typ of correlation we can specify this using the *named argument* 'method.' To run the correlation function with spearman's correlation instead we can thus add the argument method to the function call and specify that it is spearman's that we want, like this:

```
cor(vote_share_per_district,voters_per_district,method = "spearman")
```

This tells the `cor()` function to override the default method "pearson" with "spearman" which is what we want. Notice that we have to use quotation marks, `"` around spearman. If we do not, R will try to look for an object named spearman to input as the method rather than understanding that "spearman" is a piece of text that we want this argument to take. You will learn more about this in the next section on data types and classes. We can see that when we calculate the correlation using spearman's correlation instead of pearson's correlation, the correlation drops to approximately 0.394, i.e. the relationship between vote share for the first party and the number of voters is estimated to be slightly weaker.

4.2.2 Function help files

In the example above, I simply told you that in order to change the type of correlation computed you use the 'methods' argument. But how would you know that if I had not told you? And how do you figure out which arguments are available in any given function?

You do this via the R function help files. You can access the help file for a function by writing a question mark before the function's name, i.e. `?function_name`. So to get to the help file for the `cor()` function, we simply write `?cor` in R. When you do this you should see the help file for the `cor()` function appear in the *files, plots, and help* pane. It is worth noting that some functions may share the same help file if they relate to a similar family of functions. For instance, the `cor()` function shares it help file with the `var()`, `cov()`, and `cov2cor()` functions.

[FIGURE of help file]

In this help file you will see basic information about the function, what it does, which arguments it accepts, and what the output of the function is. Each help file follows the same structure, with the following parts.

- Description: A short description of the function, or family of functions are, what they are used for, and what they output
- Usage: A set of examples of how to use the function(s) and the default values of the arguments
- Arguments: A list of all arguments available for the function and what type of objects or values these arguments should take
- Details: A detailed description of what the function does, how it works, and how different arguments affect the behavior of the function. There may also be in depth discussions about the limitations of parts of the function, or special situations which may be of interest.
- Value: Details on what the output from the function are
- Note: Additional notes added by the authors of the function
- References: Books and/or articles that implementation of the function is built upon
- See Also: a list of related function with similar functionality that may be interesting for the reader
- Examples: One or more examples of how to use the function

If we look at the arguments section of the help file for the `cor()` function, you can see that there are six arguments listed there. However, not all of these can be used in the `cor()` function, since this help file is shared between several function. If you look in the 'Usage' section, you can see that only four of the six arguments are used in `cor()`, these are `x`, `y`, `use`, and `method`. However, when we used our function we did not specify the `x` and `y` arguments. Instead, we just inputed our two vectors as the first and second arguments and it worked regardless. Why?

Well, in reality, *all* arguments in R functions are named arguments. However, if you simply input the arguments in the correct order, R will interpret the first argument you have entered to be the first argument of the function, regardless of what that argument is named. Thus, when we inputed our two vectors, `vote_share_per_district` and `voters_per_district`, R therefore assumed that these two vectors were corresponding to the first and second arguments, i.e. `x` and `y` of the function. When we added our method argument, on the other hand, we had to explicitly name this argument since method is the *fourth* rather than the *third* argument of the `cor()` function. See what happens if you now run the function `cor(vote_share_per_district,voters_per_district, "spearman")`, i.e. running the function without specifying that `"spearman"` is the `method` argument. You should get an error message which says:

```
> cor(vote_share_per_district,voters_per_district, "spearman")
Error in cor(vote_share_per_district, voters_per_district, "spearman") :
  invalid 'use' argument
```

As you can see, R interpreted `"spearman"` as the value for the third argument `use` instead of as the fourth argument `method` which is what we wanted. A good practice is therefore to always use named arguments in the function, unless you are absolutely certain of the order of the arguments.

4.3 Data types and classes

Thus far we have only worked with objects that are numbers. There are, however, a large number of different data and object types which you will encounter when learning R. You can check what the type of an object by running the function `class()` on the object.

```
> class(my_first_object)
[1] "numeric"
```

As you can see, R tells us what we already know. That the object `my_first_object` is a number, i.e. that it has the class `"numeric"`.

There are a large number of different classes in R that you will encounter, but some of the most common ones are:

- `numeric` which are simply one numeric value, or a collection of numeric values known as a *vector*
- `character` which are made up of text and are usually referred to as *strings*. These strings are always enclosed by quotation marks. An object with class `character` may also be a collection of strings, and is then known as a character vector
- `factor` which are a variable with multiple categories. These categories may be numeric or labelled (i.e. have names). In general, the classes `factor` and `character` are very similar, but behave differently in some cases. We will return to the differences between these two at a later stage.

- `matrix` which is a rectangular data structure with rows and columns. Matrices can be thought of as a collection of vectors where each vector corresponds to a single column or row. Importantly, matrices require that all the data inside the matrix are of the same type, for instance `numeric`, `character`, or `factor`
- `data.frame` or `tibble` which are similar to a matrix, but allows for different data types inside the same data frame. Data frames are the general structure for data sets where each row corresponds to an individual observation, and each column corresponds to a variable. We will dive a bit deeper into the difference between `data.frames` and `tibbles` in a later chapter.

Chapter 5

Getting help

Chapter 6

Writing for-loops and functions

X

Assignment for workshop 1

Part II

Workshop 2: Data in R

Chapter 7

Importing data

x